Synchronous message passing

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2021/2022

Base model – 1

- Client-server interaction style
- The server publishes the interface of the services that it provides
  - Typed **entry** channels (aka *entry queues*)
  - With associated `in-out` protocols
- The client makes an **entry call** naming the target server *and* the entry channel of interest
  - Providing `in` parameters as required by the service protocol
  - This corresponds to making a “send” call
- To deliver a service, the server must **accept** the entry call corresponding to the relevant channel
  - This corresponds to making a “receive” call
- Service delivery is *synchronous*
  - The server acts on the service and the client wait synchronously for the corresponding output

Base model – 2

```
task type Operator is
  entry Query (A_Person    : in Name;
               An_Address  : in Address;
               A_Number    : out Number);
end Operator;
Ann : Operator;
```

Channel entry protocol

```
task type User;
task body User is
  My_Number : Number;
begin
  …
  Ann.Query(
     "…", "…",
     My_Number);
  …
end User;
```

Entry call

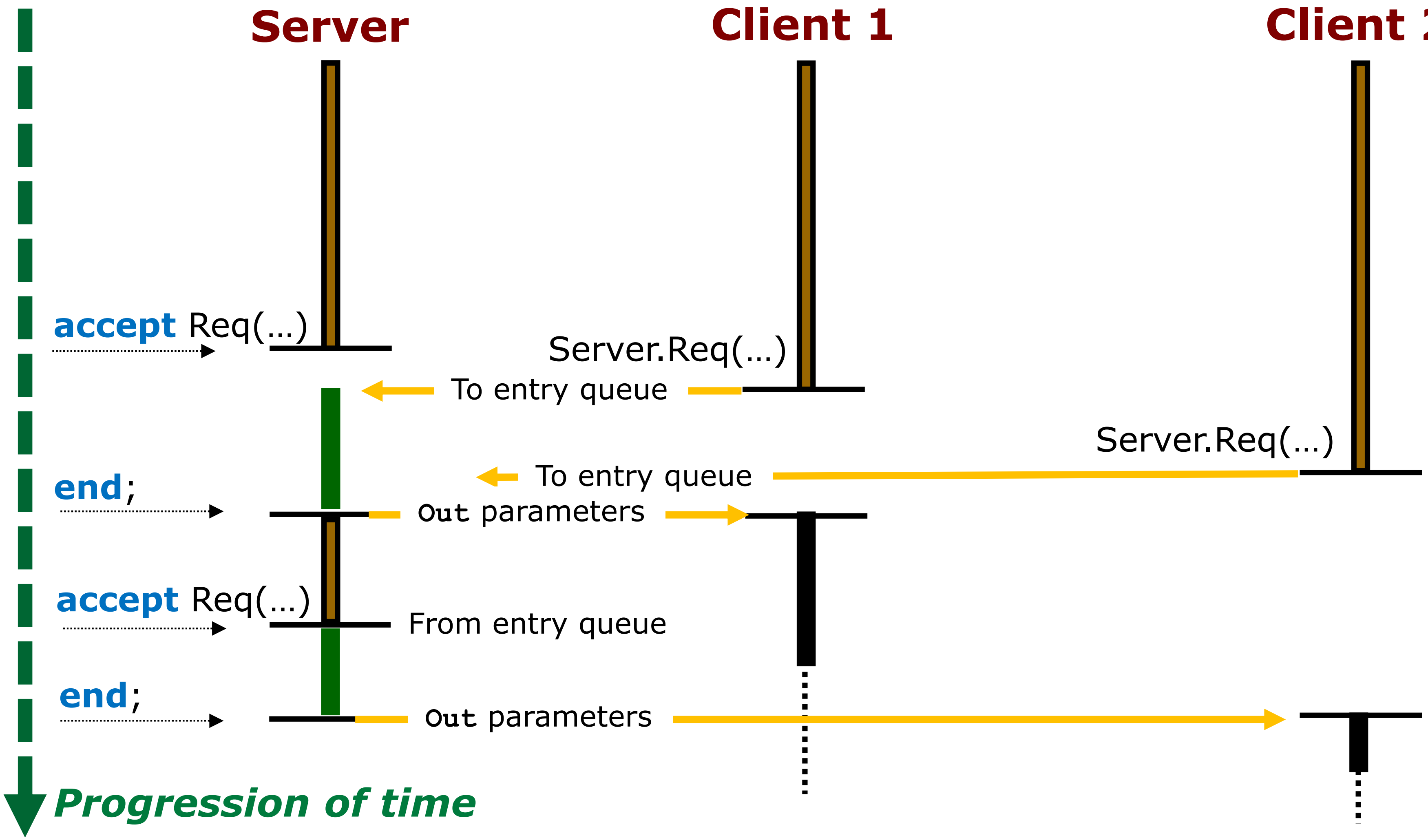
```
task body Operator is
begin
   …
   loop
    accept Query(A_Person    : in Name;
                 An_Address  : in Address;
                 A_Number    : out Number)  do
    …;
   end loop;
   …
end Operator;
```

Service implementation

Base model – 3

- Historically called *rendez-vous*
  - The client and the server meet at either side of an entry
- When the synchronization occurs, the `in` parameters flow *atomically* from client to server
  - As in a procedure call, except this is **not** a procedure call
- The server executes the service actions on the call parameters, **atomically** to the client
- At the end of the service execution, the `out` parameters flow back from the server to the client
- At that point, the synchronization ends and each party resumes their respective progress

Base model – 4



Server
Client 1
Client 2
accept Req(…)
Server.Req(…)
To entry queue
Server.Req(…)
end;
To entry queue
Out parameters
accept Req(…)
From entry queue
end;
Out parameters
Progression of time

Base model – 5

- As in any synchronization, the side that arrives first at the meeting point, waits for the other
  - The server would wait on empty **entry queues**
  - The client would deposit its entry call in the corresponding entry queue and wait for the call to end
- The default ordering in entry queues is FIFO
  - Other queuing policies might be defined
  - FIFO ordering warrants **fairness**, any other ordering is exposed to the risk of **starvation**

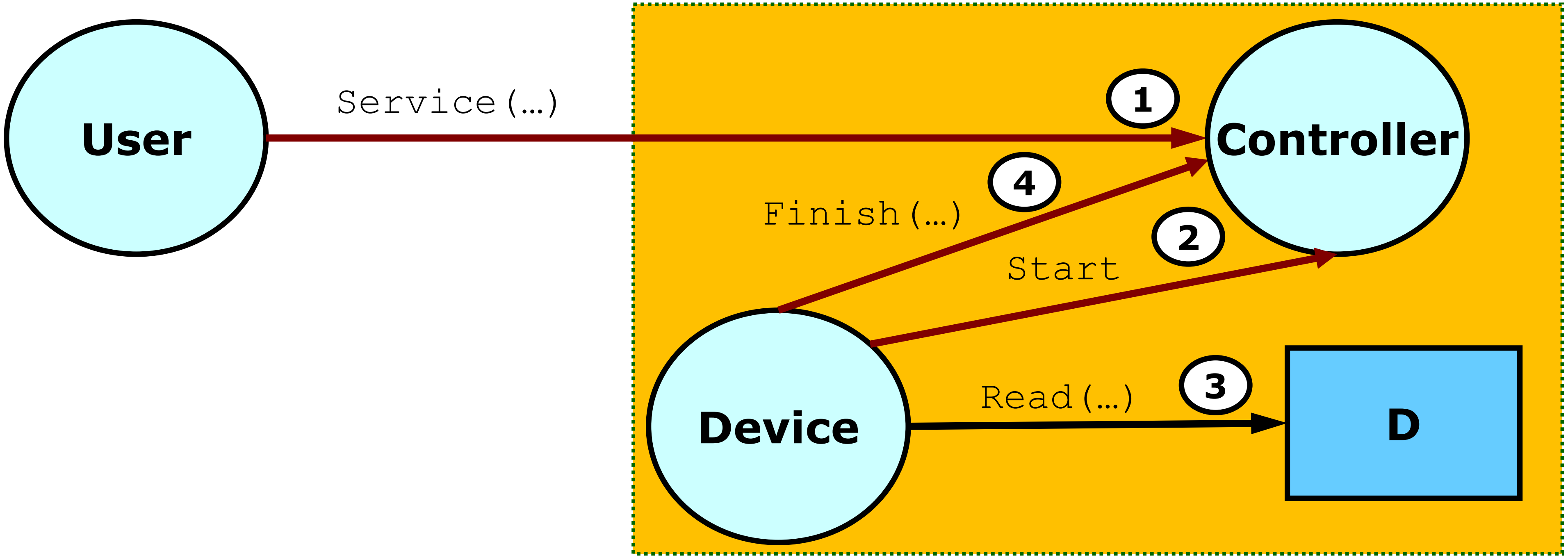
Tripartite synchronization – 1

- The *rendez-vous* model is
  - **Synchronous** for communication
  - **Asymmetric** for naming and interface provisions
  - **Bidirectional** for data flow
- During synchronization, the server is fully active and may therefore engage in synchronization with a third party
  - This opportunity gives rise to rich forms of composition

Tripartite synchronization – 2

- A server has two ways to synchronize with a third party <u>during</u> service execution
  - Embedding an entry call to another server's channel while serving its own user request
    - Thereby orchestrating a composite service delivery
  - Accepting an entry call to another of its channels
    - It must be *another* entry because the current one is locked in execution of the current service
- The latter feature (nesting accepts) requires **extending** the communication model
  - We shall discuss it next …

Nesting entry call accepts – 1



- **D** is a passive entity, accessed <u>without</u> guarantees of atomicity
- **Device** implements a state machine for commanding D, whose transitions are triggered by entry calls being accepted by **Controller**
- **Controller** encapsulates the service provided to **User** and realizes it by orchestrating its composite service protocol

Nesting entry call accepts – 2

```
task User;
task Device;
task Controller is
 entry Service (I : out Integer);
 entry Start;
 entry Finish (K : out Integer);
end Controller;
```

```
task body Controller is
begin
 loop
①  accept Service (I : out Integer) do
②   accept Start;
④   accept Finish (K : out Integer) do
     I := K;  -- azione sincronizzata
    end Finish;
  end Service;
 end loop;
end Controller;
```

```
task body User is
…
① Controller.Service (Val);
 …
end User;
```

```
task body Device is
 Val : Integer;
 procedure Read
    (I : out Integer);
begin
 loop
②  Controller.Start;
③ Read(Val); -- from D
④  Controller.Finish(Val);
 end loop;
end Device;
```

Useful model improvement – 1

- In the example, server Controller exposes *all* of its entry channels in its public interface
  - Hence, all users in the scope of that interface may have access to all of Controller's entries
  - Yet, only one of them (i.e., `Service`) belongs in Controller's service interface
    - The other two (`Start`, `Finish`) belong in the internal service logic, which should be hidden from the user
- This is a general problem
  - Service interfaces should be able to tell public entry channels apart from **private** ones

Useful model improvement – 2

```
task User;
task Controller is
 entry Service (I : out Integer);
private
 entry Start;
 entry Finish (K : out Integer);
end Controller;
```

This arrangement makes the private entry channels visible only within the internal scope of **Controller**, hence to **Device**, which is now a child task of it. Nothing changes for **User**.

```
task body Controller is
 task Device; -- nested (child) task
 task body Device is
  Val : Integer;
  procedure Read (I : out Integer) is … ;
 begin
  loop
   Controller.Start; -- child see private
   Read(Val);
   Controller.Finish(Val); -- ditto
  end loop;
 end Device;
-- continues in sidebox …
```

```
-- … continued
begin -- Controller
 loop
  accept Service (I : out Integer) do
   accept Start;
   accept Finish (K : out Integer) do
    I := K;
   end Completed;
  end Service;
 end loop;
end Controller;
```

Embedding entry calls in accepts – 1

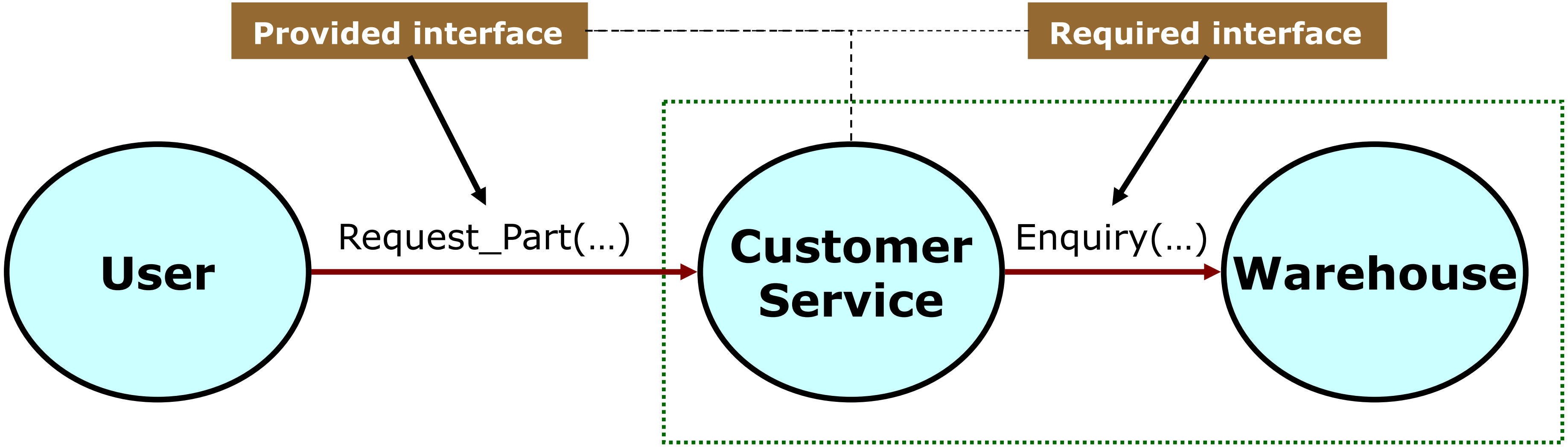
```
task Warehouse is
 entry Enquiry
   (Item  : Part_Number;
    Units : out Natural);
end Warehouse;

task Customer_Service is
 entry Request_Part
   (Part_ID  : Part_Number;
    Quantity : Positive;
    Success  : out Boolean);
end Customer_Service;
```

This solution has the defect that the service provided by **Warehouse** is publicly available while they should be private to **Customer_Service**. This defect can be fixed by normal scope encapsulation.

```
task body Customer_Service is
 In_Stock : array (…) of Boolean;
 … -- other variables as required
begin
 loop
  … -- housekeeping
  accept Request_Part
     (Part_ID  : Part_Number;
      Quantity : Positive);
      Success  : out Boolean) do
   if In_Stock(Part_ID) >= Quantity then
    Success := True;
   else
    Warehouse.Enquiry(Part_ID, In_Store);
    if In_Store > 0 then
     … -- get parts from Warehouse
     Success := True;
    else
     Success := False;
    end if;
   end if;
  end Request_Part;
 end loop;
end Customer_Service;
```

Embedding entry calls in accepts – 2



- The service interface exposed by entry `Request_Part(...)` hides the internal organization of the service delivery logic
- For this encapsulation to be correct, however, the **Warehouse** server should <u>not</u> be visible to **User**
  - This is an important design requirement
- The downside of a "server becoming client" is that its client risks a much long synchronization wait

What if …

- An exception raised during synchronization causes the *rendez-vous* to be abandoned and the exception to propagate to both sides
  - The execution incurring exception is on the server side, but the client suffers it too
- Exceptions that remain unhandled cause the master of their scope to terminate
  - That would be the case for both server and client
- Directing an entry call to a terminated server is a run-time error and causes an exception to be raised at the client side

Limits of the base model

- With the current provisions, a server can only access calls from one entry queue at a time
  - Synchronizing on an entry ties the server to that service until completion: other entry queues may have pending calls but they will be ignored …
- Sequential clients (which is the default condition of threads) can of course only issue an entry call at a time
  - But they will have to wait for as long as it takes for the server to attend to their call …

Desirable extensions – 1

- The prime extension requirements are on the server side
  1. To probe multiple entry queues *simultaneously*
     - Very natural of a true server
  2. To limit to a bounded duration the wait time on an empty entry queue
     - Equivalent to setting a **time-out**
  3. To abandon a synchronization *immediately* if the target entry queue is empty
     - Equivalent to a zero-time time-out
  4. To terminate **automatically** when no clients in the scope of the server are able to make entry calls
     - Very desirable for a true server

Commentaries

- Server-side requirements 1 and 3 directly match the implications of Dijkstra's original model of **guarded commands**
- Server-side requirements 2 and 4 have a pragmatic, implementation-oriented flavour, more than a purely algebraic one
  - However, when something abstract has "nice" properties, it may lose them altogether when we start "fixing" them to become fit for implementation
  - A synchronous communication model with time-outs may be less convenient than an asynchronous one
    - HTTP, born synchronous, is becoming increasingly asynchronous …

Actual extensions – 1

- Server-side extension requirement 1
  - Rather natural: the server's interface may publish multiple entry channels (as we just saw …)
  - The default arrangement is that all such services are equally public and have no functional nesting
  - The receive operation must probe all queues simultaneously

```
task Server is
 entry S1 (…);
 entry S2 (…);
end Server;
```

```
task body Server is
…
begin
 loop
  select
   accept S1(…) do … end S1;
  or
   accept S2(…) do … end S2;
  end select;
 end loop;
end Server;
```

Actual extensions – 2

- Semantics of extension requirement 1
  - When no entry call is enqueued in any of the server's queues at the time of evaluation, the server is put on hold on the `select` command
  - The evaluation occurs **simultaneously** for all of the entry queues in the `select` construct
  - When multiple such entry queues are non-empty, Dijkstra's model wants the choice among them to be **non-deterministic**
  - The default queuing policy for entry calls is FIFO

Actual extensions – 3

- A little refinement of server-side requirement 1
  - The entry channels should have Boolean **guards** to convey functional pre-requisite for the *order* with which entry calls should be considered for service

```
select
   Guard_1 => accept …;
or
   Guard_2 => accept …;
or
   …
or
   Guard_N => accept …;
end select;
```

Guards are Boolean expressions of the type "**when** <condition>" il
Their evaluating to True enables the **select** construct to consider the corresponding entry channel for service
All guards within a **select** construct are evaluated **once**, **simultaneously** at the beginning of that command execution

Actual extensions – 4

- Server-side extension requirements 2 and 3 aim at setting an upper bound on how long the server should wait for synchronization to happen
  - Requirement 3 wants the server to abandon the wait **immediately** if no entry call is in the queue(s)
  - Requirement 2 allows for waiting a **non-zero** time
- The runtime does different things in the two cases
  - When the wait time is non-zero, it must arm an alarm clock for that duration
  - When the wait time is zero, it need not

Implementing requirements 2 and 3

- The server may want to only consider entry queues that hold calls at the time of evaluation, doing other work if none does
  - This feature reduces the wastage of busy wait

```
select
 accept A;
or
 accept B;
else
 C;
end select;
```

The desired effect can be obtained in two alternative ways

Explicitly (preferable for zero wait)

Implicitly (with zero wait for T=0.0)

```
select
 accept A;
or
 accept B;
or
 delay T;
 C;
end select;
```

Example of use

```
task type Heartbeat_Watchdog (Minimum_Distance : Duration) is
 entry All_is_Well;
end Heartbeat_Watchdog;

task body Heartbeat_Watchdog is
 Allowable_Latency : constant Duration := …;
begin
 loop
  select
   accept All_is_Well;
   … -- client is alive and well
  or
   delay Allowable_Latency;
   … -- heartbeat may have failed, raise alarm
  end select;
 end loop;
end Heartbeat_Watchdog;
```

Dijstra’s model of guarded commands applies to time-bounded alternatives as well. Omitted guards evaluate to `True`.

Actual extensions – 5

- A server whose clients be no longer able to make calls should terminate (requirement 4)
  - As clients and servers are realized as active threads, they go about their life independently
    - However, clients must have visibility of their server if they want to make entry calls to it
    - Hence, the scope that encloses the server must also enclose its clients
  - Having the server poll for its clients is not desirable: a more general solution is required
    - Leveraging the runtime's ability to check the status of "wildlife" in the same scope as the server

Implementing requirement 4

- A `terminate` alternative can be added to the `select` construct to signify that the server should be considered "complete" when
  - Its **master** has completed its execution
  - Any other threads that depend on that same master is either terminated or suspended on a `select` command with an open `terminate` alternative
    - Clause 1 ensures that no new client can come into existence in the master's scope
    - Clause 2 applies transitively and its closure signifies that the master's scope is completely inert

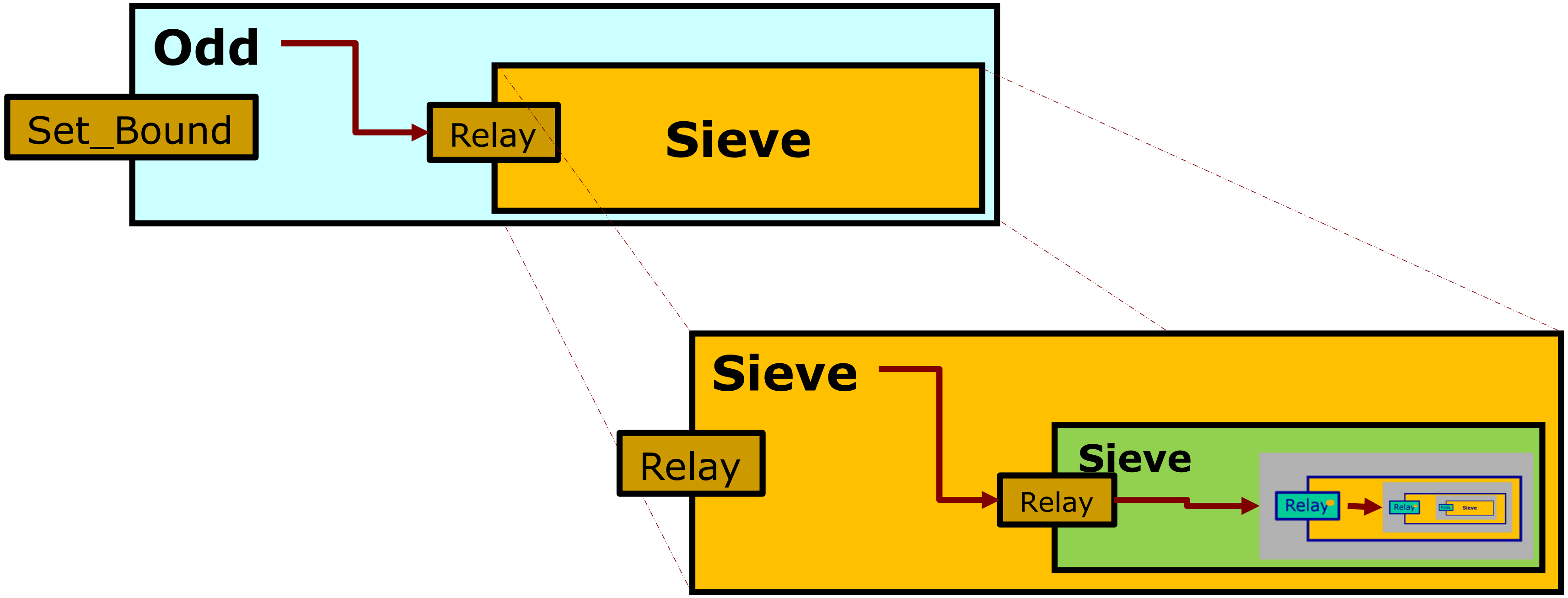
Ramifications

- The termination implied by the implementation of requirement 4 should be **graceful**
  - This requires introducing the notion of **programmable scope finalization**
- Certain extensible abstract types can be made “finalizable”
  - Their definition has an implicit abstract **`finalize`** method that the runtime must invoke when an object of that type has to cease to exist
  - Scope-based programming languages make “leave-scope” situations (**`end`**) explicit

Example of use (in exercise mode)

- **Eratosthenes' sieve**: synchronous version
  - A recursive-descent algorithm realized as a nested concurrency program in which each master-descendant pair interacts by *rendez-vous*
    - Leveraging the default FIFO queuing of entry calls
    - Leveraging the atomicity warranted by synchronization
  - We want the runtime to detect when the program should terminate and have it happen gracefully
    - We want to observe such gracefulness programmatically

Observations



- The recursive-descent nature of the algorithm transposes into hierarchical nesting of threads
  - **Odd** is the root of the hierarchy, subject to the program's main, which is its master
  - Sieve threads are all dependent, nested as shown
- The depth of recursion in the algorithm is initially unknown
  - This needs using a sentinel or the select-with-terminate construct …

Desirable extensions – 2

- The client-side extension requirements are less critical: sequential clients cannot make multiple calls simultaneously
  1. To abandon a synchronization immediately if the target server were not available instantaneously
     - Symmetrical to server-side requirement 3
  2. To limit to a bounded duration the wait time on an unattended entry channel
     - Symmetrical to server-side requirement 2

Client-server model

- A **server** is a reactive entity capable of warranting exclusion synchronization on access to its internal state
  - Idle until interrogated: no autonomous action
  - Each `accept` alternative is a critical section
  - The shared state must be private to the server

```
task body Buffer (…) is
 … -- the shared state
begin
 …
 loop
  select
   when …
    accept Put (…) do … end Put;
   … -- local housekeeping
  or
   when …
    accept Get (…) do … end Get;
   … -- local housekeeping
  or
   terminate;
  end select;
 end loop:
end Buffer;
```

```
task type Buffer (…) is
 entry Put (…);
 entry Get (…);
end Buffer;
```

Bad practice

- In addition to suffering infinite wait, the use of *rendez-vous* is also exposed to the risk of circular-wait deadlock
  - Each entry call is tantamount to a critical section protected by exclusion synchronization

```
task T1 is
 entry A;
end T1;
…
task body T1 is
begin
 T2.B;
 accept A;
end T1;
```

```
task T2 is
 entry B;
end T2;
…
task body T2 is
begin
 T1.A;
 accept B;
end T1;
```

Good practice

- Threads should be either **active** entities, capable of autonomous independent execution, or **reactive** entities, which expose entry channels for clients to invoke and synchronous communication with them
  - “Pure” servers should accept entry calls but <u>not</u> make them
  - Shared resources should be strictly encapsulated

Thread states at run time